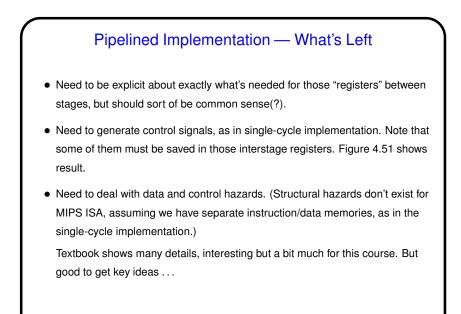


Pipelined Implementation — Some Details

• Figures 4.36 through 4.40 show some details of how this implementation works for different groups of instructions. Textbook's notation is that state elements whose right side is highlighted (blue) are being read, and those whose left side is highlighted are being written.

Slide 4

• Note that we now spot a flaw in the design: At the point where we need "which register to write to?", it's no longer correct. Figure 4.41 shows how to correct.



Slide 5

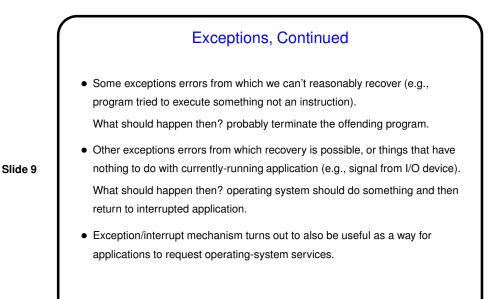
Data Hazards — Overview Some kinds of data hazards can be addressed by providing additional paths for data to flow ("forwarding"). For others, have to stall the pipeline. (Figures 4.53, 4.56.) "Stall the pipeline"? can get that effect by not changing registers or memory, and not changing program counter (so in effect the instruction being fetched is fetched again), and/or by inserting a nop instruction on the fly. Smart compilers can (at least sometimes) avoid stalls by reordering instructions.

Control Hazards — Overview

- Several ways to deal with control hazards:
- Could just stall pipeline. (Apparently not done.)
- Or could implement "delayed branches" always execute instruction after the branch. (Look at figures and confirm that this will work.) Apparently what MIPS does? (So SPIM not quite accurate implementation of ISA.) Annoying if writing assembly-language programs, but few people do, and compilers can cope?
- Still other ways (used in other architectures?) involve "flushing" in-progress instructions (before they change anything!), possibly combined with various schemes for predicting branch outcome. Details no doubt interesting, but not trivial!

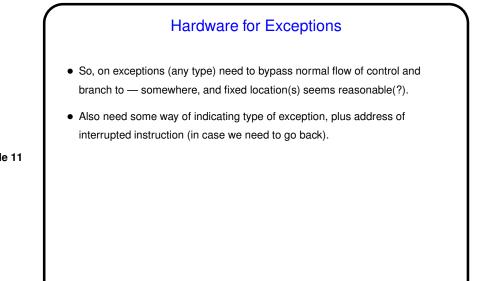
- Exceptions
 As in higher-level programming languages, situations at this level where you want to bail out of the normal flow of control because something has gone wrong (e.g., arithmetic overflow).
 Further, situations in which you want to alter normal flow of control to deal with something happening outside processor (e.g., I/O device has finished something you previously asked it to do). (You could check it periodically, yes, but usually that's inefficient.)
 Some architectures distinguish between "exceptions" (first case) and
 - "interrupts" (second case), but all kind of the same thing, so MIPS doesn't; all "exceptions".
 - What should happen on exception? Several possibilities

Slide 7



Exceptions — Hardware Versus Software

- Hardware must save current PC (with a caveat) and transfer control to fixed location(s) with an indication of cause of exception.
- Code at fixed location(s) must "do the right thing" for the exception, as described previously. Normally this code is part of operating system.
- Caveat: Pipelining complicates exception processing must allow instructions prior to the interrupted one to complete, complete or flush the interrupted one, etc. Textbook has (some of) details.
- Slide 10



Slide 11

